Bellhop Underwater Acoustic Simulation and Data Processing Optimisations: Getting Hi Def on the Desktop

Chung T C Fang (1), Anthony C Zander (1), Francesco Larizza (2), Eyad R Hassan (1) and William S P Robertson (1)

(1) School of Electrical and Mechanical Engineering, The University of Adelaide, SA, Australia
(2) Saab Australia Pty. Ltd, SA, Australia

Abstract - Bellhop is an established underwater acoustic raytracing model. Modern wrapper libraries have allowed Bellhop integration with popular environments such as Python and it remains a relevant and useful tool. However, its practical performance can be limiting when trying to simulate scenarios to capture transient interactions between multiple sources in detail. In this study we document the techniques we have developed to overcome many of these limitations.

1 INTRODUCTION

The Bellhop ray tracing platform has seen wide application in the sphere of underwater acoustics. Its diverse use cases include event reconstruction for NATO anti-submarine exercises where daily event data is simulated to identify detection opportunities for crew debriefing and enhanced training (Strode et al., 2023). In wireless underwater communication, Casari et al. (2014) described Bellhop as a component in a pipeline of oceanographic and network simulation tools to predict performance, minimising deployment effort. In research, Bellhop calculated sound field was used to establish enhanced probability of detection of submarines with multiple collaborative agents (Lan et al., 2024). References to the application of Bellhop are numerous in publications spanning from over a decade ago to the present.

The facility to simulate underwater sound propagation using Bellhop has been conveniently brought into the Python environment through wrapper libraries such as Arlpy, expediently interfacing with popular tools for machine learning and data analysis. Yet, for the most part, the libraries only act as data conduits to the core Bellhop legacy code. Developed decades ago, this code base has not kept pace with evolution of the modern PC. Naive application of the Bellhop model strictly according to documentation underutilises modern hardware. Consequently, simulation detail and productivity experience avoidable limitations.

Our particular interest was in visualising the combined sound field from many local emitters while also observing the effect in the far field. This required a dense resolution simulation over a wide expanse in area and time. The modelling requirements presented a series of challenges that extended beyond the raytracing task, including handling of large simulation data output. Initial trials demonstrated that producing a simulation for multiple sources using the naive approach quickly exhausted available memory and required prohibitive processing time. These bounds to simulation detail and productivity can be lifted by leveraging the potential of modern hardware through data parallelisation of large time series datasets, which utilises multiple CPU cores, and making use of fast non-volatile solid-state storage to judiciously free up faster volatile memory.

To achieve these processing improvements, we built upon the base functionalities provided by Arlpy, tailoring algorithmic strategy and data flow pathways for modern hardware to achieve orders of magnitude performance gains in speed and resolution. A case study is presented with a typical desktop PC, demonstrating speed and memory efficiency gains over the initial naive implementation. This paper describes the methodologies we implemented in the Bellhop Python workflow so that they may benefit other projects.

2 METHODS

In this study we use an example of two emitters separated by half wavelength horizontally with both at constant depth. The source input signal is a 1m wavelength (1.5kHz, sampled at 48kHz), 10ms pulse with counterphase to the adjacent source. Correct interference of this arrangement produces reinforcement horizontally and annulment vertically. A simplified propagation environment will be used with a uniform speed of sound profile and flat bathometry in deep water (no reflection or refraction). The aim is to demonstrate the expected combined 2D acoustic field visualised over time in a 60fps animation.

2.1 Standard Implementation

The Bellhop documentation (Porter, 2011) provides concise and approachable explanation of the software aimed at the novice user. The Python interface library Arlpy (Chitre, 2023) also provides documentation for interfacing with the Bellhop code with a modern programming environment. Closely following the steps given by these resources will provide a base (Naive) case for further comparison. Steps for simulating waves propagating and producing a time series is illustrated in Figure 1

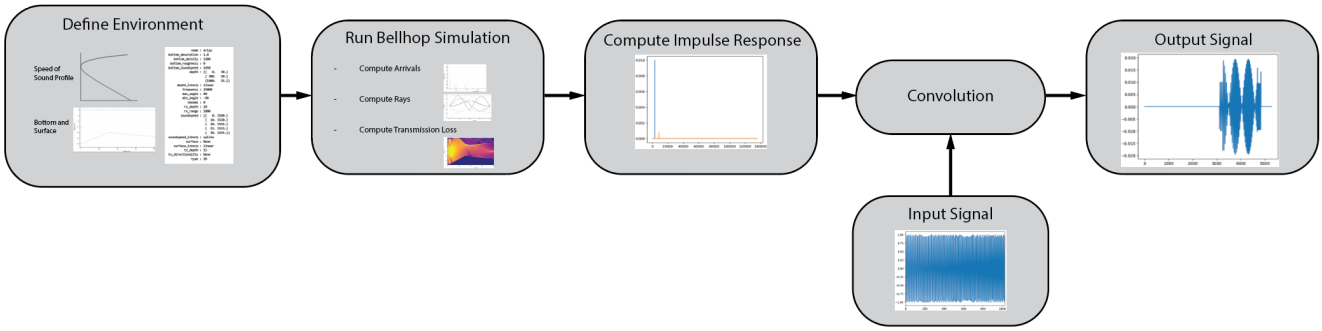

Figure 1 Standard Bellhop process as described by the User Guide.

In a 2D field case, the resulting data is a three-dimensional data block with the dimensions being depth, range and time. Wave propagation can be visualised by plotting depth versus range while progressively moving down the time axis.

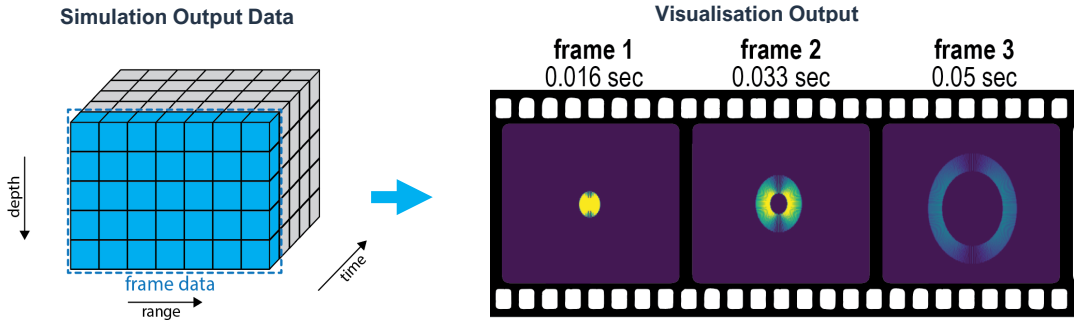

Figure 2 Desired result for visualisation are field intensity plots at regular intervals in time for frames of an animation. The simple example from Section 2.1, visualises propagation and interference between two brief counter phase pulses from sources with half wavelength horizontal separation. Output frames are combined into an 60 frames per second animation.

2.2 Simulating Multiple Sources

Bellhop simulations use a reference frame in the local coordinate system of a source. Each simulation therefore consistent of a single source in the horizontal direction, locate at range zero. For scenarios with multiple emitters, we offset the source centred ray arrivals data for each emitter to map each source from simulation half space to a global coordinate system (Figure 3). In global space each spatial element is superimposed with those from other sources to produce the combined field. In the case where sources are horizontally aligned at constant depth, emits uniformly and the bottom is flat, we can reuse a single simulation for all sources.

Issues arise when implementing source superposition directly with the Naive approach. The data size of variables in memory and processing time will grow rapidly with both resolution and the number of unique sources when ray data is converted to a time series. Improvements to enable higher resolutions will be discussed in the following sections.

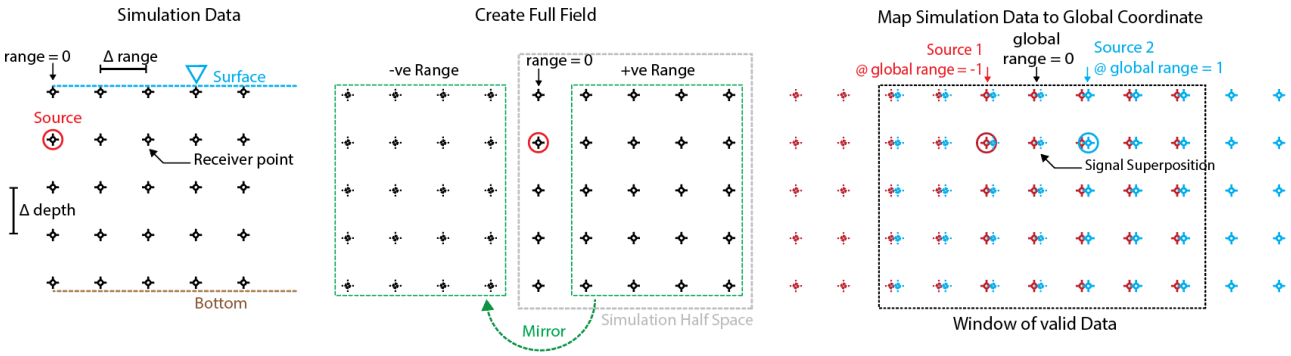

Figure 3 Constructing multi source simulation by shifting and superimposing multiple simulations in global space.

2.3 Data Parallelisation

Bellhop and Arlpy are written for single thread execution. Multicore modern CPUs can be leveraged by dividing the input data into smaller chunks, then mapping the segmented data to simultaneously executed instances of necessary functions. Finally, output fragments are gathered to recompose the full simulation result (Figure 4).

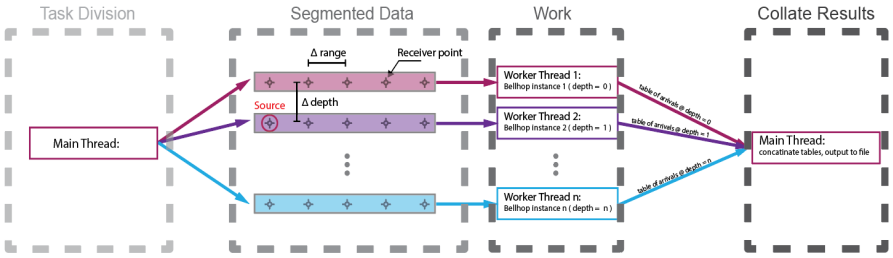

Figure 4 Simulation is forked to depth segements, 'Work' such as the Bellhop simulation, is performed on each segement concurrently by separate CPU cores. The final result is recombined to a 2D grid for output.

Given that the water surface is typically a constant fixed reference in acoustic simulations, a logical subdivision for the spatial data is along the depth axis. Each division would then consist of a row of data for all ranges at one particular depth. For uniform emitter multi-source simulation, interference between sources can be calculated by shifting the single depth array horizontally to match the offset between sources in global space (refer to Figure 3), limiting high latency disk operations.

Data parallelisation addresses certain issues of the naive implementation, as computation time is shortened by distributing the workload to multiple cores and memory consumption is reduced by subdividing the data into small chunks so that only a portion needs to reside in memory at one time. Dividing the data appropriately can minimise disk access reducing communication cost.

Using a data parallelisation approach as described follows directly from the standard single thread code and is simple to put to practice. However, there are still several shortcomings to this approach. Firstly, each process thread requires a large amount of its own quarantined memory, which limits the number of threads that can run simultaneously; secondly, a large amount of intermediate calculation data is maintained in memory before the output can be produced. Both of these issues lead to constraints on achievable resolution. The following section will address these issues by further optimising the process further.

2.4 Optimised Method

The optimised method discussed in this section addresses the resource efficiency issues of the previous section, while maintaining the ability to superimpose multiple sources as a key requirement.

The approaches taken previously can be summarised as: ray arrival data → impulse response → output by convolution → superposition (see Figure 1). Significant memory consumers are the time series data of the output signal, the impulse response and the input signal. Our strategies focus on these bulk data elements to improve efficiency, which are as follows:

(S1) Segmentation Along Time Axis: Data from the Bellhop simulation is regularly segmented along the user defined spatial dimensions (depth and range). Segmenting along the time dimensions adds complexity as there is temporal dependence that can span across the division. However, by slicing the output in time we only need to keep data relevant to that slice in memory. Segmentation in time offer the biggest impact on memory size as the time dimension is typically the largest, by orders of magnitude.

Breakdown of Key Components: ***Output Segmentation Along Time Axis***
(refer to item in Process Diagram, Figure 6)

(S1.1) Frame start times: Create an array of start times for each frame using desired frames per second up to the duration of the output. The start time of the next frame provides the end time of the current frame. Time value in seconds is needed for filtering of relevant rays from the ray arrivals table.

(S1.2) Filter ray arrivals: Gather relevant rays for the current frame and receiver location. Due to interframe dependence, rays before the current frame must be included up to the input signal length.
Note: See Implementation Note 2)

(S2) Eliminate Impulse Time Series: The impulse response is a sampled time series representation of the Bellhop produced ray arrival time and amplitude. The time series must extend for the period to the last arriving ray and generated at the rate of the input signal sampling frequency. Memory consumption can be reduced if we take the 'lazy' calculation approach, only producing the required output signal values at the time it is needed directly from ray arrivals.

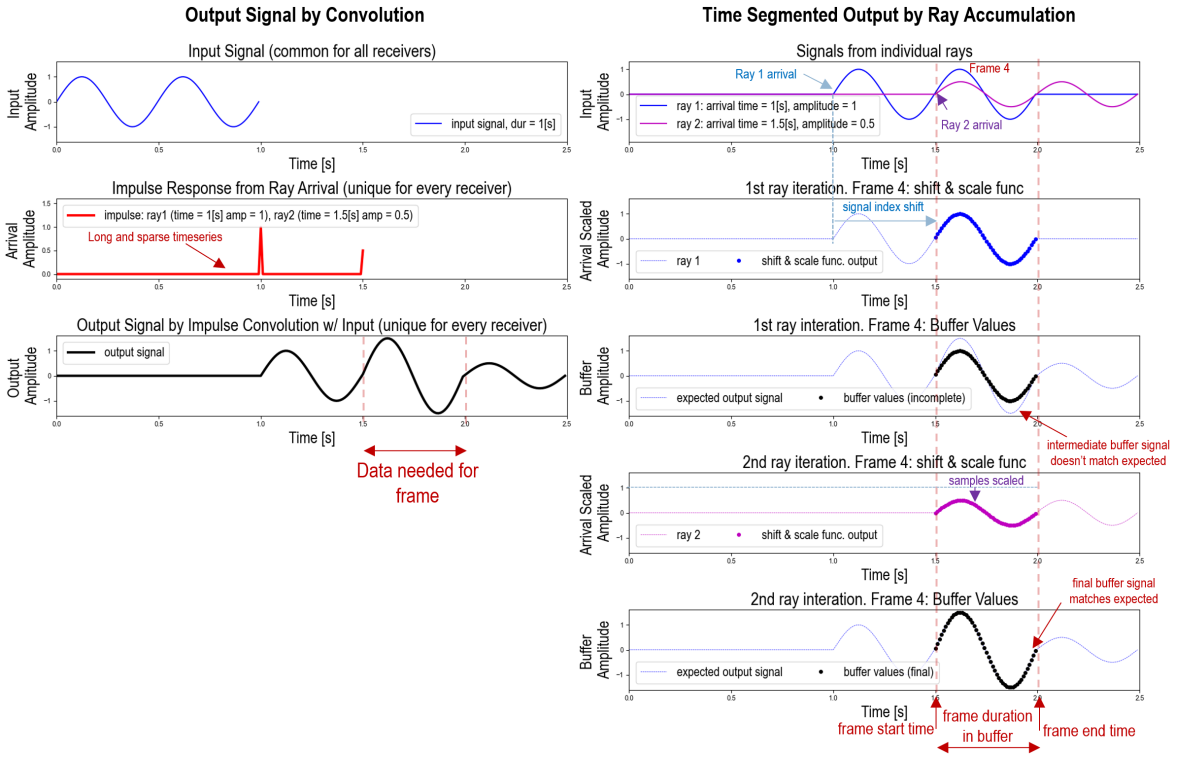

Figure 5: Output signal generation example. Convolution (left) and ray accumulation (right). Convolution requires the full signals, ray accumulation with input shift and scale is signal length independent. Note shorter sequence length.

*Breakdown of Key Tasks**: Eliminate Impulse Time Series***
(refer to item in Process Diagram, Figure 6)

(S2.1) Inter-frame buffer: Each thread holds an array spanning the inter-frame duration where each element is a sample of the combined signal. Initialised with zeros, this buffer substitutes for the convolution result and enables superposition between incoming signals as each ray is processed to build up a final signal at the current snapshot in time.

(S2.2) Generate single ray signal: Produces an array of signal values for the inter-frame period for a single ray. The required signal can be constructed with operations:

1. Subtract the ray arrival time with the frame start time.
2. If time difference is positive, front zero pad buffer up to time difference
3. If time difference is zero or negative, (where the ray arrived in a previous frame), return values of the input signal starting from the time difference.
4. End zero pad up to the remaining length of the buffer.
5. To save calculation time, if the ray arrival time plus the input signal length is less than the frame start time or arrival time is after the frame end time, the ray makes no contribution to the current frame, skip to the next ray.

These steps can manipulate the array index and do not require the full time series.

(S2.3) Scale and Accumulate ray signals: To superimpose signals between rays, update the inter-frame buffer by adding its current values with the shifted signal from S2.2 scaled by arrival amplitude of the current ray.

(S2.4) Down sample interframe buffer: After iterating all rays from all sources, the inter-frame buffer will hold the combined output signal for the frame duration. As we are only interested in a single value representing the frame, the buffer sequence can be summarised by taking its root mean square or another appropriate down sampling method, resulting in a single value output to the output frame buffer (S3.1).

(S3) Shared Memory: Sharing memory blocks between all threads for common data such as the input signals and the ray arrivals table will eliminate duplication, reducing memory consumption.

*Breakdown of Key Components: **Shared Memory***
(refer to item in Process Diagram, Figure 6)

(S3.1) Current Frame Buffer: 2D array representing the spatial field grid (range vs depth). Each worker thread writes its result to the single element (pixel). Upon completion by all workers, this array will be returned as output by the main thread.

(S3.2) Ray Arrival Table: Bellhop table of rays. Relevant fields are "*rx_range*", "*rx_depth*" (coordinates of the receiver), "*arrival_time*" and "*arrival_amplitude*".

(S3.3) Source global coordinates: 1D array of range offsets needed to shift from Bellhop's source centred local coordinate to the global coordinate.

(S3.4) Input Signals: Array of time series representing the input signal at each source.

(S3.5) Other data: Other required data such as sampling frequency, have inconsequential size and assumed to be passed to threads via function inputs.

(S4) Atomic Work Division: Previously, division was made along depth increments so that each process calculated an entire row of range pixels. Leveraging the above strategies, task division can be performed 'atomically' for the smallest logical unit of work, a single element of all dimensions of the output, i.e., space (pixel) and time (frame). Thread count memory limitation is mostly eliminated, increasing parallelisation potential. However, lower frame rates require more memory due to longer inter-frame buffer; this can be resolved by initially outputting at higher frame rates then downsampling frames.

Breakdown of Key Components: ***Atomic Work Division***
(refer to item in Process Diagram, Figure 6)

(S4.1) Compose worker inputs: Division of labour among workers occurs via individual input parameters for each worker. In the spatial dimension a worker is allocated a unique pixel using the simulation receiver grid divisions in the global coordinate. In the time dimension every worker is given the same start and end time. Other required data is also passed via the initial inputs such as shared memory location and sampling frequency of the input.

(S4.2) Map task to workers and start threads: The action of feeding a numerous list of jobs to a limited number of hardware cores is conveniently managed by thread pool libraries. In Python the '*multiprocessing.pool.starmap()'* function is appropriate as it allows multiple inputs to be mapped to instances of a common function. '*starmap()*' is blocking until all workers complete thus simply handling synchronisation. A non-blocking version '*starmap_async()*' returns a futures object which can be queried for completion status. Here the main thread can continue to do useful work while waiting for workers to complete, such as writing previous outputs to disk. For conciseness the simpler blocking implementation will be illustrated.

2.5 Initiation and Coordination

The process flow of implementing the aforementioned strategies and components from above is conceptually illustrated in Figure 6. Numerical prefix refers to relevant key components described previously. Further detail can be found in the commented code at the link provided.

Entry Point: Starting from the user's upstream script, relevant data is collated for passing to the processing script. An appropriate approach might be using a name, value pair datatype, such as a Python '*dictionary'* which can be serialised and saved to file with the '`pickle`' library). Location of the input file itself can be provided via command line arguments when calling the script.

Input Elements: Configuration and data provided by upstream process:

1. Total Time (i.e. end time) of the output.
2. Frames per second of the output; maximum being the sampling frequency of the input signal, minimum determined by the volatile memory capacity of the system.
3. Global Coordinates [source $_n$]; list of range locations for each source within the global space.
4. Ray Arrivals [source $_n$]; list of ray arrivals for each source (simulation output from Bellhop)
5. Input Signals [source $_n$]; list of input signals for each source.
6. Sampling frequency of the input signals

Main Thread: Manages stages of computation from input to output. Composed of two sections, initialisations followed by looping each frame until all outputs have been produced. The general sequence is: loading user inputs, make preparations for workers by load relevant data to shared memory, divide task for each worker thread, start worker threads, coordinate synchronisation and collates returned results for output.

Worker Thread: Once started, this code runs independently until assigned task is complete. Starting from a coordinator function with input parameters provided by the main thread, then calls various helper functions to complete each stage of the task.

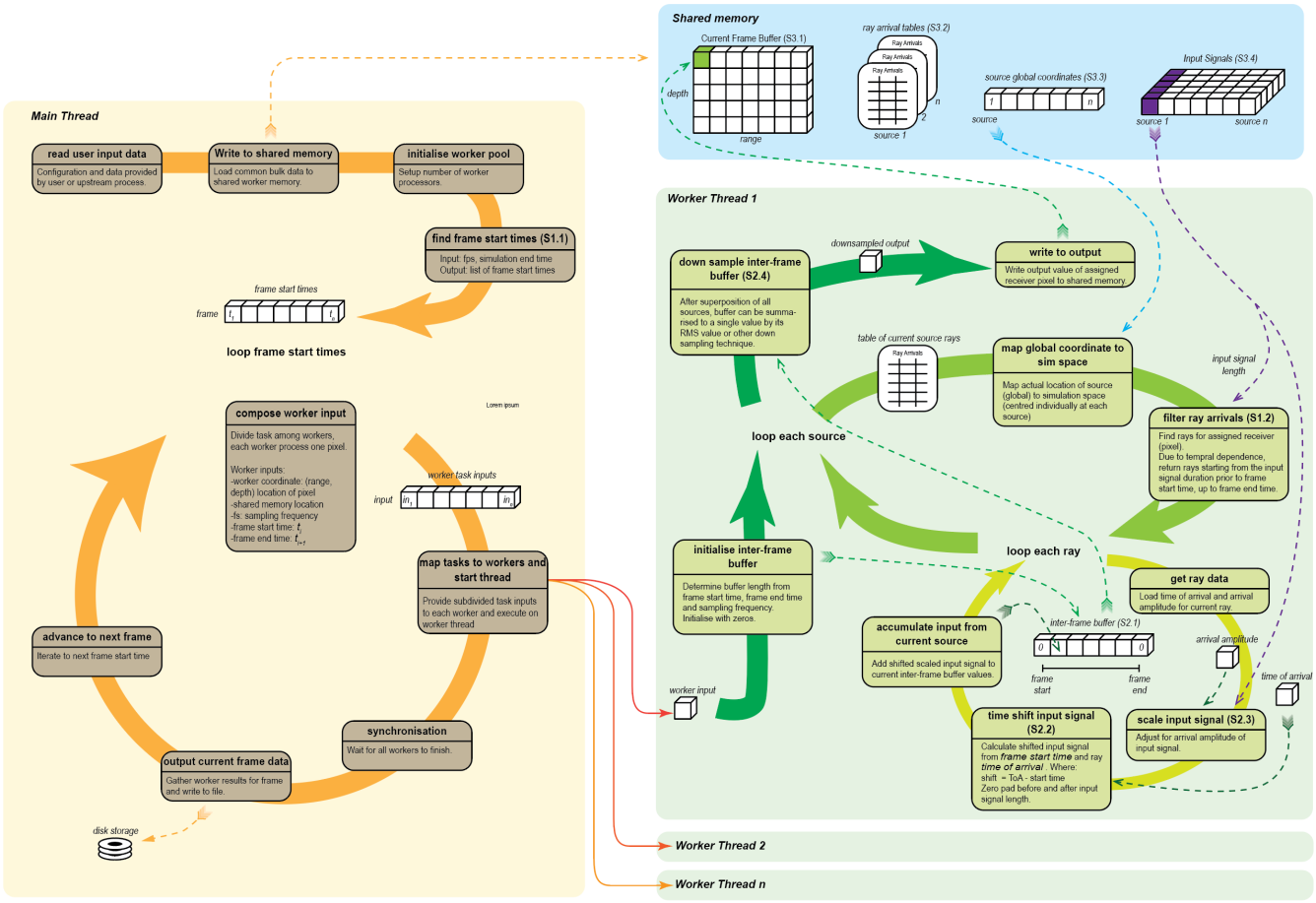

Figure 6 Process flow diagram. See numerical references in the Breakdown of Key Components for more detail about each task.

Implementation Notes: We encountered some subtleties worth highlighting:

1) On Windows the Python multiprocessing library does not work in the Notebook environment. Calls to the library must be placed in a separate '.py' script file. This script can be called within Notebooks with the '`%run`' magic command (e.g. `%run my_script.py --input $my_input_str`). Path to the input configuration data file can be passed to the script as a command line argument (the `argpass` library is useful). ' `$` ' prefix expands Notebook defined variables to their string values.

2) When using Bellhop with Arlpy, the precision of coordinates is only 3 decimal places. Appropriate rounding is required for search.

3) `multiprocessing.shared_memory` library is available for Python V3.8 (2019) or later

4) Bellhop arrival amplitudes are provided as complex numbers (datetype: `numpy.cdouble`). For correct interference between sampled timeseries inputs, only use the magnitude of the amplitude. Correct phase shift in this case is accounted by spatial separation between sources.

3 RESULTS AND ANALYSIS

Performance of the three implementations were assessed with two interfering acoustic sources at three spatial resolutions (5k, 125k and 2M total 'pixels'). All model configurations were kept the same as described in Section 2.1. The test system was a recent generation office workstation running Windows 11 on a 24 core CPU (Intel i9 13900K), with approximately 50GB of free RAM and M.2 disk storage (7GB/s write). On this PC, the naive

implementation could only complete the 5k test, while the depth segmented parallel implementation completed the 125k test but not the 2M test. The optimised algorithm completed all three. The following sections will outline the improvements in memory and speed.

3.1 Memory

For each implementation, the size of all variables in volatile memory for all threads combined is presented in Figure 7. These are grouped in the three resolution scenarios and demonstrate size growth with resolution. All algorithms increase memory usage with resolution. Naive approach memory usage grows fastest since it maintains the entire dataset in memory, expanding consumption along dimensions for all space, time and the number of sources. The direct parallelisation approach improves the situation by dividing data along the depth dimension; it also takes algorithmic shortcuts by accumulating the final output from each source, thus eliminating two dimensions of growth. However, it introduces an additional dimension of growth for the number of threads. The optimised method makes further algorithmic improvements, eliminating the large impulse response, along with use of shared memory. This enables work to be divided for a single element of the output, reducing input data size and data duplication per thread.

Exhaustion of volatile memory was the limitation in each case preventing test completion. Naive and direct parallel algorithms both failed when reaching the same memory size boundary. The optimised approach divided data to units along dimensions for range, depth, time and number of sources making the algorithm several orders of magnitude more lightweight for the same number of threads, thus maintaining stability for increasing values along these dimensions. Controlling memory size was the key attribute which enabled higher resolution outputs.

Note that some variables such as ray arrival tables and input signals have negligible size for the test scenarios and are not considered, but may be significant depending on use case.

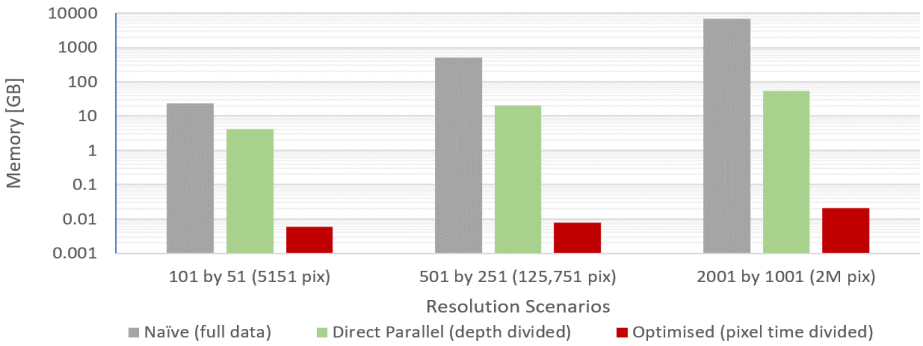

Figure 7 Total memory footprint for each implementation (memory in gigabytes is plotted on log scale).

3.2 Speed

Processing time was measured on completion of each major result component and summed to give total run time. Comparison between the three strategies is presented in Figure 8. Performance increase over the serially implemented Naive approach is given as the speed-up ratio (Figure 8 *right*). The effect of CPU core utilisation is apparent with the parallel code able to produce the final result 8.4 times faster than the single threaded code. The Bellhop simulation component was a minor contributor to total time in our scenario but benefits from a 6.9 times speed-up for the larger model. This came purely from segmenting the receiver grid and running multiple instances of the original Bellhop code on each segment. The increase in speed-up at higher resolutions is typical and due to the thread-pool overhead contributing a smaller portion of the overall computation time for larger tasks. The direct parallel implementation makes a trade-off to free up memory by writing intermediate results to high latency disk storage. The time penalty is evident at the impulse response stage where a light workload produces a large data output resulting in slower performance than the Naive implementation (speed-up < 1).

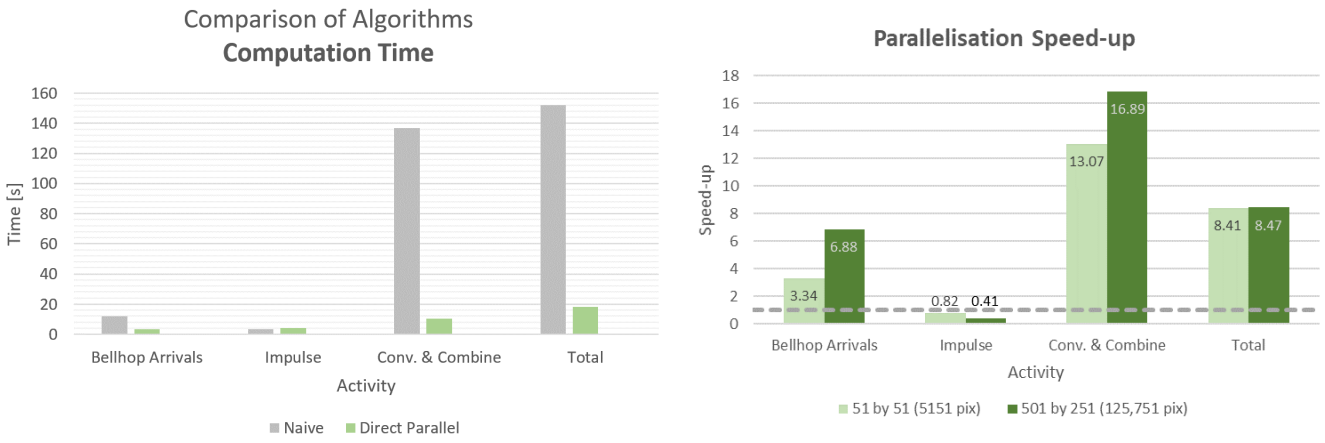

Figure 8 Algorithm speed comparison for 101 by 51 (5151 pixel) test case (left), speed-up ratio compared to single threaded Naive implementation (right)

4 SUMMARY

A naive application of Bellhop and data processing to visualise transient field interactions between multiple sources generates large datasets that overwhelms most desktop machines even at low resolutions. This article described our methods for limiting the data explosion for such scenarios, which prevented successful completion of simulations in a Naive implementation. Utilising Bellhop ray arrivals data with alternative algorithms for combining time segmented signals while leveraging parallel processing, we demonstrated substantial improvement in resolution capability and speed for producing identical results. In the test scenario our method reduced memory consumption by 5 orders of magnitude and achieved 8.4-fold speed-up over the Naive implementation.

5 ACKNOWLEDGEMENT

I would like to acknowledge this publication is supported by Saab Australia Pty Ltd, The University of Adelaide and the Australian Government Department of Education through the Trailblazer Universities Program.

6 REFERENCES

Acoustics Toolbox (2023). Heat, Light, and Sound Research, Inc. http://oalib.hlsresearch.com/AcousticsToolbox/

Casari, P., Tapparello, C., Guerra, F., Favaro, F., Calabrese, I., Toso, G., Azad, S., Masiero, R., & Zorzi, M. (2014). *Open source suites for underwater networking: WOSS and DESERT underwater*. IEEE Network, 28(5), 38–46. https://doi.org/10.1109/MNET.2014.6915438

Chitre, M. (2023). *Underwater Acoustics.* ARL Python Tools Documentation. https://arlpy.readthedocs.io/en/latest/uwa.html

Lan, C, Yu, Z, Chen, H, Zhang, L & Zhang, M. (2024), '*Research on Underwater Collaborative Detection Method Based on Complex Marine Environment*', IEEE Access, vol. 12, pp. 1–1. https://doi.org/10.1109/ACCESS.2023.3348782

Mbambo, T. (N.D). *Argparse Tutorial*, Python Software Foundation. https://docs.python.org/3.11/howto/argparse.html

multiprocessing — Process-based parallelism (2024). Python Software Foundation. https://docs.python.org/3/library/multiprocessing.html

multiprocessing.shared_memory — Shared memory for direct access across processes (2024), Python Software Foundation. https://docs.python.org/3/library/multiprocessing.shared_memory.html

Porter, M. (2011). *The BELLHOP Manual and User's Guide*. Heat, Light, and Sound Research, Inc. http://oalib.hlsresearch.com/Rays/HLS-2010-1.pdf

Strode, C., Akbulut, B., Been, R., & Tesei, A. (2023). *Acoustic reconstruction of ASW serials during NATO exercises*. Underwater Acoustics Conference and Exhibition (UACE) 2023, Grecotel Filoxenia, Greece. https://www.uaconferences.org/docs/2023_papers/UACE2023_2074_Tesei.pdf

7 ONLINE REPOSITORY

https://github.com/AUMAG/bellhop-parallel